Langley Grant
IN-61-CR P-70
93166

Annual Progress Report

Contract No. NAG-1-260

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Mr. Edmund H. Senn
ACD, Computer Science and Applications Branch

Submitted by:

J. C. Knight
Associate Professor

Report No. UVA/528213/CS85/106
January 1985

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

The Implementation and Use of Ada On Distributed Systems With High Reliability Requirements

Annual Progress Report

John C. Knight

Department of Applied Mathematics and Computer Science
University of Virginia
Charlottesville
Virginia, 22901

January 1985

CONTENTS

1. Introduction

The purpose of this grant is to investigate the use and implementation of Ada* in distributed environments in which reliability is the primary concern. In particular, we are concerned with the possibility that a distributed system may be programmed entirely in Ada so that the individual tasks of the system are unconcerned with which processors they are executing on, and that failures may occur in the software or underlying hardware.

Over the next decade, it is expected that many aerospace systems will use Ada as the primary implementation language. This is a logical choice because the language has been designed for embedded systems. Also, Ada has received such great care in its design and implementation that it is unlikely that there will be any practical alternative in selecting a programming language for embedded software.

The reduced cost of computer hardware and the expected advantages of distributed processing (for example, increased reliability through redundancy and greater flexibility) indicate that many aerospace computer systems will be distributed. The use of Ada and distributed systems seems like a good combination for advanced aerospace embedded systems.

During this grant reporting period our primary activities have been:

(1) Continued development and testing of our fault-tolerant Ada testbed on our DEC VAX 11/780.

* Ada is a trademark of the U.S. Department of Defense

(2) Relocation of the testbed to a network of Apollo DN300 professional workstations.

(3) Consideration of desirable language changes to allow Ada to provide useful semantics for failure.

(4) Analysis of the inadequacies of existing software fault tolerance strategies.

(5) The preparation of various papers and presentations.

A summary of the various implementation activities of our fault-tolerant Ada testbed are described in section 2. The sequencer has been given a new, relatively sophisticated control language, and it is described in section 3. A major part of our effort has been the relocation of the testbed to the Apollo network, and this activity is described in detail in section 4.

In our analysis of the deficiencies of Ada, it has been quite natural to consider what changes could be made to Ada to allow it to have adequate semantics for handling failure. In section 5, we describe some thoughts on this matter reflecting what we consider to be the minimal changes that should be incorporated into Ada. These thought are included in a programming language that is a variant of Ada that we call *Ada 2*. The design of Ada 2 is fairly complete and will appear in a PhD dissertation shortly [1]. We include a summary of some aspects of Ada 2 in section 5.

We consider it to be important that attention be paid to software fault tolerance as well as hardware fault tolerance. The reliability of a system depends on the correct operation of the software as well as the hardware. Software fault tolerance is rarely used in practice and when it is used, it is ad hoc with no formalism or organization. One of the reasons for this state of affairs is the general inadequacy of existing proposals for building software in a fault-tolerant manner. Before reviewing Ada and trying to incorporate software fault tolerance mechanisms into the language changes we consider necessary, we have reviewed the state of the art and prepared a systematic set of criticisms of existing proposal for the provision of fault tolerance in software. This set of criticisms is summarized in section 6. These criticisms have lead us to develop a new language facility for backward error recovery called the *colloquy*. This construct is briefly introduced in section 5 but is described in detail in Appendix 2.

During the grant reporting period we have made various reports about this work. Our activities in this area are described in section 7. Appendix 1 contains an example program that has been run on the testbed. Appendix 2 contains a paper about our work in software fault tolerance that has been submitted to the Fifteenth Symposium on Fault-tolerant Computing to be held in Michigan in June.

2. Implementation Status

We have continued our implementation activities of both the testbed and the associated translator. The translator translates a subset of Ada which includes most of the tasking and exception handling mechanisms into code for the virtual processors implemented by the testbed.

Some parts of the testbed have had to be redesigned and reimplemented as a result of obtaining a more accurate understanding of the way in which Ada operates. In many cases, the language definition is very obscure and it is quite difficult to determine exactly what is meant. In other cases, the semantics are comprehensible but extremely complex making an accurate implementation difficult. An area that has given us a great deal of difficulty is the exception mechanism. It appears relatively simple and straight forward as first but the many possibilities for exception generation during processing of declarations for example makes an accurate implementation very difficult. Our implementation of the exception mechanism has been redesigned and the implementation has been revised.

The overall state of the implementation can be gauged from the fact that the simple program that we have used as an example in various papers and presentations has been successfully executed using the translator and the testbed. The source text of the program that was executed is contained in appendix one of this report. A small number of other tests have been run and used to find errors in the translator and testbed. We are just beginning a systematic effort to debug the system.

The system continues to run on a single VAX using UNIX processes to simulate computers and UNIX pipes to simulate communications facilities. We had intended to use a network of IBM Personal Computers as the target of this testbed. The use of the VAX/UNIX combination has always been viewed as an interim step that allowed us to develop the software in a relatively convenient and friendly environment. Clearly the facilities of the IBM PC are relatively limited although probably adequate with sufficient care. The major problem of porting the testbed to the IBM PC's would be the very long compile times resulting from the slow processor, the small memories, and the use of floppy disks.

Our department has been fortunate in receiving funds for the purchase of some Apollo workstations. At the time of writing, the department has ten workstations connected together via a token-ring bus, and the network also has a 300 Megabyte disk system. They are equipped with relatively large main memories, and in general are more powerful computers than the IBM PC's. We feel the Apollos are more appropriate for the support of the testbed. The Apollos also support a variant of UNIX which makes them somewhat compatible with the testbed as developed on the VAX.

In order to avoid spending inordinate amounts of time investigating the idiosyncrasies of the Apollo system or building pieces of support software, we decided to wait until other research projects had successfully used the Apollos and demonstrated that they could provide the facilities we need before we attempted to use them. An early effort to use the Apollos [2] showed that they could provide Pascal support and communications' support but the speed of communication was very low. The reason for the slow speed is the approach used by the Apollos for user-originated inter-node communication. All such traffic has to go through a disk-based mail box with the result that the transmission speed is disk limited. We decided that the perfrormance was adequate given the other inefficiencies of our testbed and proceeded to move the entire system to the Apollo network. That transfer has been completed and the system is now operational on the Apollos. The details of the process involved in moving the system to the Apollos are described in section 4.

As well as replacing the control language for the sequencer, we have replaced the interface that each physical processor provides to the experimenter. Recall that each physical processor supports an arbitrary number of abstract processors, and that each abstract processor supports an arbitrary number of virtual processors (Ada tasks). In the VAX implementation, each physical processor is actually a UNIX process but it is equipped with a terminal which appears to be an operator's console. On the Apollo network, each physical processor is actually a DN300 workstation which is equipped with a monitor. To allow the experimenter to keep track of the activities that are under way on each physical processor, we have implemented a series of displays that the experimenter can arrange to be displayed on the operator's console for the physical processor of interest. Each of these displays is updated as execution proceeds and displays are provided to show:

(1) the overall status of the physical processor,

(2) the status of each abstract processor on that physical processor,

(3) the status of each virtual processor on that physical processor,

(4) all the current breakpoints for all the virtual processors on that physical processor,

(5) the status of all the simulated I/O devices on that physical processor.

These displays will be extended and enhanced as we discover what information is most interesting to the experimenter. Even now however, we find the information very useful, and, for example, can show the way in which the abstract processor's time is being multiplexed among the virtual processors.

3. Sequencer Control Language

Recall that the testbed is trying to allow experimenters to answer "what if..." questions about concurrent Ada programs. The sequencer control language is the experimenters interface with the testbed and so its form and facilities are extremely important.

Why is control of parallel programs any different from sequential programs? The reason is that "what if..." questions about tasking cannot be answered easily (sometimes never) because, in most implementations, a set of tasks cannot be forced into the necessary state that leads to the "what if..." question. This is not the case with sequential languages because they are deterministic. In most debugging systems for sequential languages there is a single-step facility whereby effects of individual instructions within a program can be studied in detail. Concurrent languages, on the other hand, are nondeterministic. There is no guarantee that a particular state of interest is reached on any given execution. For example, suppose a set of Ada tasks is executing asynchronously on the Ada testbed with the scheduler controlling which task runs when. The experimenter may be interested in asking questions such as: "What would happen if this particular task were forced into a certain state in its execution and this other task were forced to stop at a specific point in its execution?" and then "What do the contents of memory look like for a particular virtual processor at this point?". These questions are typical of those asked for controlling parallel programs. This is the level of control that is essential for the monitoring and experimentation of these Ada tasks. Hence, the main function of the command language is to provide the facilities for performing this control. Control is needed not only to single-step individual tasks, but to single-step them in relation to each other.

The command language interpreter provides the interface between the user command level and the sequencer module of the testbed. It receives the command line, interprets it, and passes the validated information to the rest of the sequencer which is then responsible for actually performing the actions to carry out these commands.

In the design of the sequencer command language, there are basically two elements essential to the design for control of Ada tasks. They are the ability to monitor, in some meaningful way, the tasking activity so as to understand the behavior of the parallel tasks, and the ability to perform experiments based, either implicitly or explicitly on the information gathered. Through the interaction of these two elements, the user can attempt to gain an understanding of the causes of existent errors or at least to note where the implementation and the expected behavior of the parallel tasks differ.

The overall strategy that is taken in the design of the command language is to control Ada tasks, not to debug Ada programs. First, the testbed must be viewed from an operational semantic definition standpoint: semantic in that it pertains to answering questions of language meaning; operational in that it allows programs to be executed and their actions to be observed. Furthermore, the definition must provide the ability to answer the "what if..." questions

Given these general requirements, we established the following minimal set of detailed requirements for control of the sequencer and hence the testbed:

(1) Starting a desired experiment. This requires the availability of the compiled Ada code to be interpreted and the map showing how the abstract processors for the experiment are to be mapped to physical processors.

(2) Executing named tasks. This requires a list of the task names (any number) that the experimenter wishes to start executing. This command was originally separate but it has been included with the command for restarting tasks which have been stopped. This was done since the involved tasks are each at their own fixed code location and the one command for starting could then be viewed as a set of tasks being suspended at a particular breakpoint (breakpointing being the ability to temporarily halt an executing program); for the initial starting up of a task's execution then this breakpoint would be defined at location zero. The start would always be from a current breakpoint.

(3) Exiting from the existing test environment. A provision must be made to allow the experimenter to have a summary of important system information listed upon exit.

(4) Stopping or artificially suspending named tasks no matter what they are doing. As with starting task execution, a list of the tasks, again any number, the user wishes to stop or suspend must be given. A common example of a situation that would use this command would be one in which there was the desire to observe temporary suspension of all but one process in order to eliminate interference from any of the other processes.

(5) Causing a particular abstract processor (AP) to fail. Since a major point of the testbed is to see if software strategies can tolerate processor failures, the experimenter should be provided with the ability to fail any processor. Giving the AP number of the particular AP to be failed would cause the physical processor owning the subject AP to cease to schedule it.

(6) Setting and unsetting breakpoints. The general problem regarding breakpoints involves the desire to have tasks suspended in the middle of statements. Since AP code may be shared among tasks, specification of breakpoints by location only is insufficient. Therefore, a breakpoint has to be defined such that it is named by the source-level task name (task id) and a code location. It is also considered desirable that the effects of a breakpoint be delayed so that a task must execute that code location more than once before "hitting" the breakpoint. This latter facility is required to provide more flexibility to the user and his desire to perform experiments with loops or end conditions

(7) Restarting tasks' executions. As described above, a list of task names would be given to start or resume any number of tasks executing. This would allow the named tasks to run until they encounter a breakpoint or terminate. The ability to restart task execution is important because many fault-tolerant strategies call for automatic replacement of defective hardware.

(8) Single stepping a particular task. This would require the name of the task that is to be involved and the number of instructions that are to be executed before the subject task is temporarily halted; absence of the count should yield a default of single stepping the named task through the interpretation of exactly one instruction. This capability would allow a user to deal with tasks through a perspective which is more microscopic than the Ada source language level. For instance, each process can be brought to the desired state by executing to a breakpoint set for that process and single stepping for fine adjustment from there.

(9) Displaying the sequencer's tables. These displays would provide a quick and useful reference of which tasks are running, where and what there current

breakpoints are, etc.

(10) Displaying the state of the testbed's data structures. This level of control would be valuable in decisions that must be made regarding branches. The user could breakpoint before the branch, display the memory contents and decide what to do next on the basis of that. All of these display capabilities would provide the means of monitoring whether the fault-tolerant strategy that is being tested works or not.

(11) Calling upon a help facility. This would permit the user at any time before, during, or after the experiment to view the available commands that are allowed; syntax and usage of each command would be provided.

(12) Recalling commands. This would allow the experimenter to look at a log of commands that he has used.

With this set of command facilities, the experimenter will have a good basis for implementing the kind of control that is needed in a first, elementary, but useful control mechanism for Ada tasks. It satisfies the two elements initially described as essential to the control of Ada tasks: it possesses commands to allow the ability to monitor the tasking activity at a microscopic level and it provides the ability at any moment of the inspection to perform experiments as to the future endeavors of those tasks. This set is by no means complete and there exists a lot of remaining issues that require investigation before further expansion of the control mechanism can be made.

Listed below are the actual commands of the command language interpreter as presently implemented:

NEW

Start an experiment. The names of the files containing the AP to PP map and program must be given.

QUIT

Exits an experiment without having a summary dump listed.

QUITD

Exits an experiment and has a summary dump listed.

RESUME

Starts or resumes any number of tasks executing; execution will stop when a breakpoint is hit. The names of the tasks to be resumed must be listed; a "*" in place of the task name list will resume all currently started tasks.

STOP

Stops any number of tasks executing. The names of the tasks to be stopped must be listed; an "*" in place of the task name list will stop all tasks that are running.

KILL

Causes one AP to be killed (failed). The AP number to be killed must be given.

BREAK

Sets a breakpoint according to a named location. The task name and address in the task at which to set the breakpoint must be given. An optional count may be given to indicate the number of times to execute the instruction before stopping occurs; the default is one.

UNSET

Unsets a breakpoint according to a named location. The task name must be given. An optional code offset may be given to indicate the address in the task at which the breakpoint was set. If no code offset is given, all breakpoints for that task are unset.

SINGLESTEP

Executes the named task one instruction at a time for the given number of instructions. The task name must be given. A count is optional to give the count of instructions to execute with the default being one.

DISPLAY APTOPPMAP

Displays the AP_number to PP_number map.

DISPLAY VPTOAPMAP

Displays the VP_name to AP_number map.

DISPLAY TASKTOVPMAP

Displays the task_id to VP_name table.

DISPLAY VPDATASTRUCTURE

Displays the VP data structure.

DISPLAY VPSTATE

Displays the state and location of the named VP.

DISPLAY BREAKPOINTS

Displays all of the breakpoints in the current experiment.

HELP

Displays all of the available commands with the ability to give a description of each.

FLASHBACK

Displays the last specified number of commands. If no number is provided it defaults to 15.

The command language interpreter also provides in its command language several other capabilities and features including abbreviations for the commands, good error handling and feedback of the error messages to the user, checks made on all parameters, a UNIX-like MORE facility for certain commands like the flashback command, and sensible screen layouts.

4. Transporting The Testbed To The Apollo Network

From the start of the project, we had attempted to keep the system portable by segregating what we thought would be machine dependencies in the programs into separate files. These are called "include files" because a compiler directive in the main source file can cause the contents of these files to be included inline. The intent was that the included files could be easily replaced by other include files whose contents would be specific to the new machine. Up until this port, the include files we had been using had been specific to the VAX under UNIX. The original substitute include files had been specific to the IBM personal computers running MS-DOS. The change of target from the IBM personal computers to the Apollo workstations necessitated the construction of include files specific to these machines and their operating system. This operation involves replacing declarations and the bodies of Pascal procedures. Some procedures had to be given null bodies. There were cases in which we had to create new procedures to duplicate the functionality of certain "standard" Pascal features which were missing in Apollo Pascal. Other procedures had included system calls and had to be reprogrammed do deal with the different interface presented by the Apollo operating system. Later in the project, we were forced to return to this step due to an upgrade in the Apollo operating system.

Not all machine or system dependencies can be removed from the main source files into include files. Most of these have to do with idiosyncrasies of the compilers involved. Examples are the syntaxes required by the different compilers for the "include" and "external" directives and the larger set of keywords recognized by Apollo's extended version of Pascal. The latter necessitated a systematic respelling of variable names. Due to the differing directory structures of UNIX and the Apollo operating system, the pathnames for files named in compiler directives had to be changed. Since copies of the machine specific include files for both the VAX and the Apollos reside on the VAX, they cannot have the same file names. We needed to change references to VAX-specific files to Apollo-specific references. Apollo Pascal requires a separate explicit "open" call following each "reset" or "rewrite". All instances had to be so modified. The VAX version had needed to use UNIX "pipes", a tool which required us to give up the standard input and output files in the interpreters and use other files to deal with the terminal interface. On the Apollos, only the standard input and output files can be used for the terminal interface. This required each occurrence of the alternate file names to be changed. On the VAX, the default size of integers is 32 bits, but Apollo Pascal uses 16 bits instead. Uses of the type "integer" were systematically changed to "integer32" in case the variable being declared needed that much room. Some do.

We wanted to continue to maintain a single version of the system into which we could incorporate any future enhancements and which could be automatically modified to bring those enhancements to any targeted machine. The principal instance of the testbed is the VAX version. In order to make enhancements to the VAX version automatically available to the Apollo version, all of the modifications described above needed to be automated. Thus we built *filters* (programs that transform program text) to accomplish these changes in a systematic way and a *shell script* (a set of commands written in the UNIX command language) to effect the transfer of files from the VAX to the Apollos. Finally, the parser tables for the user interface (command interpreter) are represented in binary. We had to create a pair of filters to convert these tables to text for the transfer across machines and to convert them back into the binary format suitable to the Apollos.

This system was apparently one of the first large set of files to be transferred over the communications path between the VAX and the Apollos. Some of our files would get across and others would not, and those which did get across often had their contents altered which was only detected during attempts at compilation. We had to make several attempts before we got all of the files transferred properly.

Our attempts at compiling the system on the Apollos pointed out the errors in transferring the source and some minor undocumented differences of the target compiler. Most Pascal implementations which do not perform packing ignore the "packed" keyword. Apollo Pascal insists that it not occur. When we obtained a successful compilation, we began to try to exercise those parts of the system which required no Ada program to interpret.

During these tests, we found and corrected several items we had previously overlooked. Despite our conversion programs for transferring them from one machine to the other, the parser tables needed a different binary format than we had given them. The filters had been incomplete in that they had not caught all of the file names which neeed changing, and in that they had altered certain declarations so as to produce anomalous behavior at execution time.

The pre-initialization portions of the interpreter and controller main programs exist in machine-specific include files. We could not prepare much to go into these files until we had experimented with initializing and operating the Apollo communications mechanism. Similarly, the machine specific files having to do with communication between interpreters and the controller in the testbed contained calls to routines which had not been built.

In the interest of time and effort, we intended to reuse some low level routines written (in C) for another researcher's project. These routines were to provide access to the Apollo inter-node communications facilities, and a more easily understood and used interface between our system and the Apollo operating system. These routines turned out to be wholly unusable. We found that they were misusing the Apollo primitives and almost always overwrote received messages before returning to our code. It was about this time that the aforementioned upgrade to the Apollo operating system occurred. Rather than try to fix the borrowed routines, we determined to write our own (in Pascal).

Although the Apollos communicate among themselves in a ring network, all user programmed communication must pass through a *single* user-written process. This makes the Apollos resemble a star network. We had to write a program to serve as the hub of this star network. There were also certain global or "own" variables which had to be maintained for the communications routines to operate properly. This necessitated the addition of another machine dependent include file to the system. Some badly documented features of the Apollo operating system hindered progress in building the communications interface. An example is the status codes returned by system calls. The returned status code turns out to be a record and one must check different fields of that record depending on what code was returned. In other words, one must know what the returned status is in order to determine which field to find its value. This caused some tests to show that messages were not transmitted even though they had obviously been received. Determining what the real problems were and finding ways to get around them involved writing and running several programs other than the testbed system.

Once the required software was written and communication was established, the tested's user interface needed to be repaired. UNIX and the Apollo operating system have very different views of terminals, in particular of "raw mode". Under UNIX, raw mode routines resemble interrupt handlers and there is the option of detecting whether anything has been typed before being forced to read it in the Pascal sense. The Apollos, on the other hand, do everything through a screen manager which allows the user program to poll the keyboard but forces a read if anything has been typed. We had to write routines which make read-without-lookahead look like read-with-lookahead. In the process of repairing the user interface, we discovered other problems. As an example, some features of the interface were written as part of a student project. Rather than use the parser which we were repairing, each of these features included its own. We had to track down and individually repair each of these.

The files resulting from the port were transferred back to the VAX as the archival and future enhancement site. The version obtained through all of the modifications we had made was compared to the original VAX version. The differences were largely in the files which were intended to be machine specific. Certain changes, however, had pointed out portions of the system which had been machine specific despite our early efforts. These portions were moved into the appropriate files and the filters were run again to ensure that the new VAX version could be transformed automatically into the Apollo version.

In conclusion, the entire testbed has been successfully moved to the Apollo network and modified to operate there. It has run a set of elementary tests using several Apollo computers and we are convinced that it is as operational as the VAX version. The permanent version of the source code that is stored on the VAX has been modified to include all the changes necessitated by the Apollo system.

5. Ada And Hardware Fault Tolerance

Nothing is stated in the Ada Language Reference Manual about how programs are to proceed when a processor is lost in a distributed system although the manual does specifically include distributed computers as valid targets. We have summarized our concerns about Ada's inability to deal with processor failure by pointing out that the problem is basically one of omitted semantics. In particular it was found that:

(1) Although tasks could be affected by the failure of a processor which contained some context used by the task or some process it was communicating with, the language did not specify what the effect should be. In other words the *failure semantics* for Ada are incomplete.

(2) It was not clear what program units could be distributed, what the semantics of distribution were, or what the syntax for specifing distribution was. The *distribution semantics* for Ada are missing.

(3) There were many problems with communication between tasks. Not only could a task be suspended indefinitely if it was communication with a task on a processor which had failed, but timed and conditional entry calls did not provide a task with the assurance that it could eventually continue.

We have proposed additional semantics to deal with this situation. The heart of these additional semantics is the notion that the loss of a processor and consequently the loss of part of the program can be viewed as equivalent to the execution of abort statements on the lost tasks. Thus in all cases, failure semantics would be equivalent to the semantics of abort.

We have also proposed a comprehensive mechanism for implementing these semantics. This mechanism requires quite extensive changes to the execution-time support for Ada but it is feasible as we have shown in our testbed implementation.

The use of abort semantics is not the most elegant approach. There are numerous consequences that seem rather extreme if considered out of context. For example, abort semantics imply that all the dependent tasks of a task that is lost must be terminated even if they are still executing on non-failed computers. The *overwhelming* advantage of abort semantics is that they do not require that the language be changed.

A more elegant and clearly preferable approach in the long run is to modify the language and to introduce language structures that include appropriate failure semantics. During the grant reporting period we have been considering what form these language structures might take.

Although Ada ignores this problem, other languages do not and language designers have proposed various schemes in the literature. For example, Liskov has proposed "guardians" [3], and "atomic actions" [4] have been proposed by several people. We have considered both, along with other schemes, as candidates for inclusion in Ada. None of these proposals seem appropriate however because they are not able to provide the performance level that is required in the kind of applications for which Ada is intended. The naive introduction of atomic actions into Ada would reduce performance substantially; probably making the language worthless.

Given that language structures with more sophisticated semantics probably cannot be added to Ada, we have considered what more modest changes could be made that would be in the spirit of the language but would provide acceptable performance. We have broken the lack of failure semantics in Ada into two parts and addressed each separately. The two parts are *entrapment in communication* and *loss of context*, both of which we have documented extensively in the past.

Entrapment in communication can be dealt with in a revised language much like it is with abort semantics. Raising an exception in a task that is the subject of entrapment is a reasonable way to inform the task of the problem and to provide a mechanism to allow it to proceed. The difficulty that follows from something like this is the subsequent difficult with redirection of communication. Given that a task has been lost and cannot be used in further communication, it is necessary to communicate with its alternate. Since Ada (as presently defined) requires that the caller explicitly use the name of the callee in a rendezvous, a different call must be used for the alternate. This means that all communication must be guarded (probably by an IF statement) so that different entry calls can be made. This is a large burden to put on the programmer, and it can hardly be described as elegant. We have no well-defined suggestions on preferable language structures at this time.

We also observe that the Ada rendezvous makes no provision for broadcast messages. There are plenty of occasions when a single task needs to communicate with a whole set of other tasks; for example starting a set of real-time services or informing a set of tasks about machine failure at the level of the application software. This seems like a serious omission.

The loss of context problem is actually far more serious. With abort semantics, loss of context requires that parts of the program be removed when this may not be strictly necessary. One solution that we have considered is to require that the general nesting structure of the program be reflected in the way tasks are assigned to processors. For example, only tasks at the outermost level would be the subject of controlled distribution. All nested tasks would be required to be assigned to the same processor as their parent. This seems like a reasonable solution since any loss of context takes with it all the objects that could reasonably use that context. It is however a major restriction on the forms that programs may take.

The key problem with this type of limitation is that it may not be suitable at all for certain applications. Consider for example a system which includes a special-purpose hardware processor; a fast-fourier transform unit for example. The Ada code which provides access to the services of this unit will obviously reside on the unit. The fast-fourier transform functions may be required from many parts of the program but the programmer might be reluctant to make these routines global. Good programming practice may well dictate that such routines be nested. Allowing nested objects to be distributed seems almost mandatory.

In considering this problem we have concluded that it really is essential to be able to locate nested objects separately from their parents. To solve the resulting loss of context problem, we propose that Ada's scope rules be enhanced to include objects that are distributable and have limited scope. We propose that the objects to be distributed be a new form of package and that the scope of objects in the package be limited to that package only. Access to the package would be through the objects made visible in the specification of the package in the usual way.

Our consideration of this topic is not complete. We will continue to look at desirable extensions to Ada and complete the definition of the enhanced communications mechanism and the distributable packages. We include below some our thoughts at this time about how Ada might be modified. We are defining a revised version of Ada that includes all of these ideas called *Ada 2*.

5.1. Failure Semantics

Failure semantics consist of two parts. First those tasks that can be affected by the loss of another task must be specified. Second the effect of the loss on them must be given. In the previous solution abort semantics were extended to cover the case of loss of a task by processor failure. This meant that dependents (in the Ada sense) were affected; the effect was to abort all affected tasks. These semantics ignore the important differences between aborting a task which can be thought of as an effort to remove a useless part of a program and losing a task by failure where the aim is to preserve as much of what remains as possible. In fact, a special case had to be made for the main program, otherwise the loss of the main program would result in aborting all non-library tasks.

Attempts to produce less destructive semantics run up against the problem that the context usable by a program unit in Ada includes all enclosing scopes and cannot be restricted. Further no distinction is made between run-time and compile-time declarations so that a task which uses a package containing only type definitions will depend on that package even though the package may not exist at run-time.

Any attempt to restrict visibility, say by using import/export lists, runs into the problem of avoiding restriction to an incomplete set of declarations. An example will illustrate the problem:

```
type PAIR is array (1 .. 2) of INTEGER;

type LIST is array (1 .. 100) of PAIR;
```

LIST should not occur in an import list unless PAIR does. As import lists get long it becomes difficult to avoid such omissions.

Both long import lists and incomplete import lists can be avoided by encapsulating all declarations in declarative groups, each containing a complete list of declarations. The declarations contained within a declarative group are available within the begin-end part of the immediately enclosing unit. Declarative groups can be named or un-named. A named declarative group can be made available to another declarative group by mentioning its name in a with clause. An un-named declarative group cannot be made available in this way, consequently all declarations in an un-named declarative group are for strictly local use only.

If a declarative group A is named in a with clause for a declarative group B then the declarations in A must be mentioned in B before they can be used in B; remember that the declarations in B must be complete. This can be done in two ways:

(1) A declaration from A can be referenced in B by mentioning its complete name in B; for example, A.name.

(2) The name of A followed by a semi-colon is equivalent to listing all the declarations in A.

In Ada the role of a declarative group is one of the roles that can be assumed by a package; it seems better to separate declarative group which never ends up as a run-time entity (though the things declared in it might) from package which will normally represent a run-time object. Nevertheless, to avoid having to surround every package with a declarative group, it is convenient to treat a package similarly to a declarative group, by requiring that the visible part be a complete set of declarations and that access to the visible part be obtained by mentioning the package name in a with clause.

Returning to failure semantics, the first step is to define the context of a program unit to be the set of declarative groups and packages that it mentions in its with clauses.

A declarative group (package) is said to be location-restricting if it declares anything other than types and tasks. Similarly a package is said to be location-restricting if its visible part declares anything other than types and tasks.

A program unit is location-free if none of the declarative groups and packages in its context are location-free.

Below, only location-free program units will be allowed to be distributed; if a program unit is not location-free its location is determined by the location of its context. If the constituents of its context have different locations the program is erroneous. A consequence of this is that the loss of a processor cannot effect the context of a task at run-time, no action needs to be taken in this case. The other possibility is the case where a task is affected by the loss of a task that it is trying to communicate with. This is dealt with in section below.

5.2. Distribution Semantics

In the earlier work it was assumed that the only program unit which could be distributed was the task. In fact, a much better case could be make for distributing packages and, as tasks can always be encapsulated into packages, it is a more general notion. Certainly if a task is to be called, the caller and the server must have some common environment; the place for this to be specified is in the visible part of a package (remember this must be a complete set of declarations), which contains the server task.

Here it is important to carefully distinguish the information needed to create a task, the information needed to use a task, and an implementation of the task. In Ada if a task object is visible so that the task can be used then the type definition will also be visible so that a copy of the task can be created. Although the task can be hidden by using a package and renaming the entry calls as procedures, the distinction between creation and use is so important that it is worthwhile to express it more directly in the syntax of the language. The situation with the implementation is even more striking. It is clear that although the functionality of a task is defined by a single implementation, (i.e. by its body) bottom up and top-down design both work because the functionality is easier to grasp than the implementation. That being the case it would seem natural to let the same task have several bodies in particular it might be desirable to implement a task in different ways on different machines; using different speed/memory trade-offs for example.

5.3. Communication

In a distributed system where processors can fail, every communication with another processor may fail. It follows that unless a task is prepared to wait for a reply that will never come the task must take precautions. What the task would like to be able to do is to ensure that no matter what actions are taken by the communication tasks this task will be able to proceed. This can be done by having a time-out mechanism, measuring the time waiting for the reply. Of course, with such a mechanism the timeout could come too soon and the task could proceed before the reply arrived. When the reply did arrive it would have to be discarded.

It would then be possible for a task to be on *several* queues at the same time. Since this can happen by accident, it might be desirable to let a task put itself onto several queues anyway, as in the Intel surrogate call.

Further, if a caller must be prepared for the server to fail at any time, it would cause no further hardship for the caller if the server were allowed to manipulate the entry queues. In particular, a server should be able to take a call off the queue, examine it, and if necessary return it to the queue.

Essentially the system would become a data-gram service where requests for service would be done as well as possible but nothing would be guaranteed. With this understood, tasks could control their own destiny and no action need be taken for tasks affected by the failure of a task that they were communicating with.

6. Ada And Software Fault Tolerance

We have examined the literature on fault-tolerant software with the goal of determining the adequacy of Ada in providing a software fault tolerance mechanism. We find that Ada makes *no provision whatsoever* for software fault tolerance. Consequently we have considered what extensions to Ada might be desirable to support fault-tolerant software.

In examining the literature we have concluded that the schemes that have been proposed are inadequate in general and in many cases incomplete. In this section we review the inadequacies of previous work in software fault tolerance.

A general consideration for crucial systems is time. Boolean acceptance tests and voting codes must be reached and reached on time for the results to be useful at all. A common problem, which we refer to as *the unexpected delay problem*, is that some unanticipated circumstance, e.g. an infinite loop, may cause a particular section of code to be executed too late for its results to be useful or not to be executed at all. If a scheme does not address the unexpected delay problem, then it is insufficient for providing software fault tolerance in a real-time program since a program in that context needs only to be late to be considered faulty. Another consideration for a fault-tolerance scheme is the management of complexity. If the use of a scheme involves too much effort on the designer's (programmer's) part, it may be counter-productive in that more faults will be generated through the use of the scheme than would otherwise occur. Furthermore, a fault in the application of a fault-tolerance scheme might make the system more dangerous than if fault tolerance efforts had not been applied at all. A scheme supported by a rigid, encasing, structured syntax allows design-time (compile-time) enforcement of the accompanying semantic rules. Such a quality in a scheme allows for added complexity without added faults.

6.1. Exceptions

Although claimed to be suitable for software fault tolerance, exception handling can only deal effectively with anticipated faults, not the unanticipated faults addressed by an actual fault-tolerance approach. A crucial system should have anticipated faults removed before it is placed into service. Exceptions can be used within systems to represent and deal with expected, normal, but unusual situations.

In most languages, but particularly in Ada, when an exception handler is entered there is no indication of exactly from where control transferred. Neither is there an indication of how much of the state has been damaged. These problems make it difficult for a handler either to repair the fault and transfer back to the point where the exception was raised, or to replace the execution of the remainder of the "procedure".

Often the finite list of available exception names (even when user defined names are included) is very general, such as in Ada: range_check, numeric_error, constraint_error, and tasking_error. As a result, the exception could have been raised in any of many statements (components), or in one of many places in one statement. Consider, for example, the following statement:

```
I := A(J) + B(K) + C(L) + D(M);
```

If the execution of this statement raises a subscript error, there are four different subscript that could be involved. Also note that the subscript violation is a *symptom* of the actual fault. The actual fault might lie in the calculation of J or K or L or M, or it might be in some decision computation that erroneously directed control to this statement. Further, attempts to determine the extent of the damage by examining values in the state could raise another exception. Since one fault existed in the routine covered by the handler, it cannot be assumed that no others will exist in a continuation that attempts the same algorithm. Since multiple faults may have existed in that part of the routine already executed, ascribing the erroneous state detected to one fault and "handling" that one may not correct the state at all. Indeed, if the fault to which the detected error is ascribed is not one of the actual faults in the routine, the actions of the handler may cause even more damage.

Exception handling involves predicting or enumerating the faults that may occur in a system so a handler can be provided for each. This may be impracticable in a complex system. A failure to predict an exception and provide a handler for it could bring about the collapse of the entire control system or at the very least wreak havoc within some part of it. If a handler is provided for an exception with the expectation that that exception was only to be raised in one portion of a routine, but it was actually raised in another portion or propagated up from a component routine, the actions of the handler could be entirely inappropriate.

6.2. N-Version Programming

Although the method employs parallelism, it still implements software fault tolerance in logically sequential parts of a system: It is not a concept or construct for dealing with parallel programs.

The n-version programming proposals all assume that all versions will arrive at the cross-check points – they ignore the unexpected delay or infinite loop problem.

The proponents of n-version programming claim that the scheme is inherently more reliable than, say, recovery blocks. The reliability of the scheme depends upon the reliability of the voting criteria and test for agreement. That is just as volatile as the recovery block's acceptance test. How to actually do the voting is unspecified. There are discussions of different choices for dealing with single numerical values, such as weighted sums, but not for the general case of a vector of values of differing types. The discussions on voting on single numerical results concludes that that is very difficult, but most applications are going to need long vectors of results of differing types. It would seem that voting in an actual control system might become impossible. The proponents have admitted that n-version programming may not be applicable in many situations [5].

The n-version programming strategy depends upon the ability to create independent versions or programs derived from the same specification. As for how the independence of versions is to be achieved, there are appeals to the use of independent programming teams using different languages. Problems may arise from common programming experience and current fashions in algorithms, or even from a specification that specifies too much.

As for the use of different programming languages and translators, that can be a source for faults. Translators for different programming languages are likely to use incompatible representations for even the simplest data structures, and will certainly provide incompatible synchronization mechanisms. The software that attempts to rectify these differences in preparation for distribution of inputs and gathering and voting upon results, either becomes a bottleneck subject to single-point failure or must itself be made fault-tolerant. If that software is made fault-tolerant by n-version programming, the software providing the same service for it comes into question, ad infinitum.

Implementing an n-version program is not as easy as the descriptions make it out to be. It appears at first easy to do n-version programming in Ada – just put each version in its own task and let them execute. But problems arise in obtaining the results in order to vote on them and even in ensuring that all or most versions even reach the cross-check points! Infinite loop problems can occur, and arranging for a faulty task to consent to a rendezvous with the driver is no mean feat. Voting in general presents a centralized bottleneck and is therefore undesirable for distributed applications.

6.3. Recovery Blocks

Since the recovery block concept relies on syntactic support from the programming language in use, and Ada fails to provide this syntax, recovery blocks cannot be used in Ada as presently defined. However, there are fundamental technical problems with recovery blocks also and we review them in this section.

In a recovery block, there is only one test for acceptability of results. How to program the acceptance test to be both meaningful and allow a wide range of alternate algorithms to pass it is unspecified. Design diversity in the primary and the alternates, combined with the possibility of degraded service from the alternates, implies that the acceptance test must not be made very strict. It must be possible for any results of the primary or any alternate (assuming they are correct) to pass the test, yet it must be strict enough to detect errors produced by any of the primary or the alternates. This combination may not be possible. A test that is general enough to pass all valid results might not be specific enough to actually detect all errors within the construct. The strategies involved in the primary and in the many alternates may be so divergent as to require separate checks on the operation of each "try" as well as an overall check for acceptability as regards the goal of the statement. The recovery block really needs multiple tests, one for the primary and one specific to each of the alternate algorithms, perhaps with a general overall test as a check on the various individual tests.

Like n-version programming, the recovery block scheme depends upon the generation of independent versions of software, in this case, to be used as the primary and alternates. Due to the degraded service concept, the alternates do not have to produce results so close as to be able to vote upon them, but they also need a certain degree of independence to reduce the possibility that they will contain the same or very similar faults. How to get independent versions for alternates is not really addressed in the recovery block proposals.

The recovery block is strictly a sequential programming construct. It gives no hint about recovery after inter-process communication. The conversation concept is an appropriation of the recovery block concept, not an integral part.

There is the question of when a recovery block should be used. There is little indication as to what portions of a program should be protected by recovery blocks. If used on every routine and every statement sequence, the tests may become trivial and fail to offer any benefit. If recovery blocks are only used at the outermost levels, the acceptance tests may be so complex as to duplicate the complexity of the primary or alternates. This may introduce more faults in the acceptance test than the primary alone, or it may squander processing resources so that execution of an alternate would bring about a timing failure.

The infinite loop problem and its generalization, the timing of control program activities has remained unaddressed by the recovery block scheme.

How can we rectify the use of unrecoverable objects with the backward recovery strategy? There is some discussion in the literature on how recovery blocks could be reconciled with nested recovery block commitment to unrecoverable objects.

The problem of the latency intervals for fault detection being longer than commitment intervals is not addressed. That is related to the problem of how to construct meaningful acceptance tests. It is assumed that acceptance tests can be constructed that can detect errors before they become so wide spread, or that multiple layers of nested recovery blocks' acceptance tests can together detect them. The possibility of nested recovery blocks allowing such errors to "escape" should not be permitted.

6.4. Conversations

As with recovery blocks, the use of conversations requires programming language support. Again, Ada fails to provide any but this is not too surprising since there are no satisfactory proposals in the literature. This is one of the major shortcomings of conversations.

Conversations have been criticized in the past for failing to provide a mechanism preventing "desertion". Desertion is the failure of a process to enter a conversation when other processes expect its presence. Whether the process will never enter the conversation, is simply late, or enters the conversation only to take too long or never arrive at the acceptance test(s), does not matter to the others if they have deadlines to meet, as is likely in a crucial system. Thus, desertion is another form of what we have called the infinite loop problem. The processes in a conversation must be extricated if the conversation begins to take too long. Each process may have its own view of how long it is willing to wait, especially since processes may enter a conversation asynchronously. Also, a deserter can be considered erroneous, but determining which process is a deserter could be difficult. Only the concurrent recovery block scheme even addresses the desertion problem. The solution there is to enclose the entirety of each participating process within the conversation. Not only can a process fail to arrive at a conversation, it cannot exist outside of the conversation.

The original conversation proposal made no mention of what was to be done if the processes ran out of alternates. Two presumptions may be made: that the retries proceed indefinitely, which is inappropriate for a real-time system, or that an error is to be automatically detected in each of the processes, as is assumed in all of the proposed conversation syntaxes. What the syntactic proposals do not address is that, when a process fails in a primary attempt at communication with one group of processes to achieve its goal, it may want to attempt to communicate with an entirely different group as an alternate strategy for achieving that goal. This is the kind of divergent strategy alluded to above. The name-linked recovery block and the conversation monitor schemes do not mention whether it is an error for different processes to make different numbers of attempts at communicating. Although they may assume that is covered under the desertion issue, that may not necessarily be true if processes are allowed to converse with alternate groups.

Russell's work [6] permitting the application to have direct control over establishment, restoration, and discard of recovery points has its own set of problems. First of all, his premise ignores the possibility that the information within a message can contaminate a process' state. When the receiver of a message is rolled back, he merely replaces the same message on the message queue. This was the main "advantage" derived from knowing the direction of message transmission. His application area is that of producer-consumer systems. The control systems we are considering are feedback systems. A producer almost always wants to be informed about the effects of the product, and a consumer almost always wants to have some influence over what it will be consuming in the future. The relationships between sensors and a control system and between a control system and actuators can be viewed as pure producer-consumer relationships, but sensors and actuators are more accurately modeled as unrecoverable objects. The scheme allows completely unstructured application of the MARK, RESTORE, and PURGE primitives. This fact, along with the complicated semantics of conversations, which they are provided to create, affords the designer much more opportunity to introduce faults into the software system.

All of Kim's proposals [7] use monitors for inter-process communication. In a distributed system, monitors and any other form of shared variables are vulnerable to extensive delays. While a monitor may be implemented as a fully-replicated distributed database, most other implementations leave its information vulnerable to processor failure. With an independently executing process, as one would simulate a monitor in Ada, the application could decide upon appropriate times to save copies for use by a replacement after reconfiguration. But the traditional monitor is not active and long periods may pass without any process calling a procedure that updates a replacement monitor's state.

Since the name-linked recovery block proposal makes no mention of the method of communication among processes within a conversation, it remains open to charges of permitting smuggling. If processes use monitors, message buffers, of ordinary shared variables, other processes can easily "reach in" to examine or change values while a conversation is in progress. Kim also states that ensuring proper nesting of name-linked recovery blocks is impossible.

The conversation monitor is designed to prevent smuggling but, as Kim's description stands, it allows a problem that is even more insidious than smuggling. A monitor used within a conversation is initialized for each use of the conversation, but not for each attempt within a conversation. This allows partial results from the primary or a previous alternate to survive state restoration within the individual processes. Since such information is in all probability erroneous, it is likely to contaminate the states within this and all subsequent alternates.

Our conclusion from all of this is that Ada makes no provision for fault-tolerant software but that none of the proposed technologies are really complete and ready for use. Extensive work is needed to complete the theory before practical use can be made in Ada and similar programming languages. We have taken these various issues and defined a new programming construct for backward error recovery called the *colloquy*. The colloquy is presented as an extension to Ada. In our opinion, all the deficiencies of previous proposals have been solved by the colloquy, and it includes all previous solutions as special cases. The colloquy has been written up in a paper which we have submitted to the Fifteenth International Symposium on Fault-tolerant Computing. A copy of the paper is included in Appendix 2 of this report. We are presently implementing the colloquy in our Ada testbed.

7. Professional Activities

During the grant reporting period we have prepared several papers and made various presentations about this work.

In May, we were invited to a workshop sponsored by Westinghouse Space and Electronics Center in Baltimore Maryland. The purpose of the workshop was to allow Westinghouse personnel to become familiar with various technologies for crucial systems, and to expose researchers to the present and pending DoD-related projects requiring very high reliability.

We were also invited to participate in a panel session at the Distributed Processing conference held in San Francisco in May. This panel addressed distributed Ada and the other panel members were David Fisher from Gensoft Corporation, Robert Firth from Tartan Laboratories, Bryce Barton from Hughes Aircraft, and Dennis Cornhill from Honeywell. There was some agreement among the panelists and substantial disagreement. Nothing that was said affected our position on the inadequacies of Ada for distributed computing.

In the 1983annual report for this grant we included copies of two papers that had been submitted to the Fourteenth Fault-Tolerant Computing Systems Symposium (FTCS 14). One of those papers (appendix 3 in that report) was rejected. We disagree with many of the comments made by one of the referees and have written to the conference organizers requesting clarification. The second paper (appendix 4 in the report) was accepted and was presented at FTCS 14.

A lengthy paper describing most of our work on Ada in some detail was being prepared when we submitted our 1983 annual report. A preliminary version of that paper was included in that report as appendix 5. That paper has been completed and submitted to the IEEE Transactions on Software Engineering. After ten months, we are still awaiting an editorial decision from that journal. A shortened version of that paper was sent to AdaLETTERS, the publication of the ACM Special Interest Group on Ada, and appeared in volume IV, issue 3.

REFERENCES

(1) J.I.A. Urquhart, "On Languages for Programming Crucial Real-time Applications on Distributed Systems", Ph.D. Dissertation, University of Virginia, May 1985.

(2) J.N. Scott, "A Testbed for Distributed Operating System Development", M.S. Thesis, University of Virginia, September, 1984.

(3) B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust Distributed Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3.

(4) S. K. Shrivastava, "Structuring Distributed Systems for Recoverability and Crash Resistance," *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 4.

(5) L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," *Digest of Papers FTCS-8*,

(6) D. L. Russell and M. J. Tiedman, "Multiprocess Recovery Using Conversations," *Digest of Papers FTCS-9*, June 1979.

(7) K. H. Kim, "Approaches to Mechanization of the Conversation Scheme Based On Monitors," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 3.

APPENDIX 1

```
procedure EXAMPLE is

   task CALLER is
      pragma distribute_to(1);
      pragma priority(1);
   end CALLER;


   task SERVER is
      entry E;
      pragma distribute_to(2);
      pragma priority(1);
   end SERVER;

   task ALTERNATE_SERVER is
      entry ABNORMAL_START;
      entry E;
      pragma distribute_to(1);
      pragma priority(1);
   end ALTERNATE_SERVER;

   task body CALLER is
      SYSTEM_STATE : integer;
   begin
      SYSTEM_STATE := 1;
      write(1,1);
      loop
         MAIN_BLOCK:
         begin
            if SYSTEM_STATE = 1 then
               write(1,2);
               SERVER.E;
               write(1,3);
            else
               write(1,4);
               ALTERNATE_SERVER.E;
               write(1,5);
            end if;
         exception
            when TASKING_ERROR=>
              SYSTEM_STATE := 2;     -- abnormal
         end MAIN_BLOCK;
      end loop;
   end CALLER;
```

```
task body ALTERNATE_SERVER is
begin
   write(2,1);
   accept ABNORMAL_START;
   loop
      write(2,2);
      accept E;
      write(2,3);
   end loop;
end ALTERNATE_SERVER;


task RECONFIGURE_1 is
   entry FAILURE(WHICH : in integer);
   pragma distribute_to(1);
   pragma priority(2);
   for FAILURE use at 10;
end RECONFIGURE_1;

task body RECONFIGURE_1 is
begin
   loop
      write(3,1);
      accept FAILURE(WHICH : in integer) do
         write(3,2);
         ALTERNATE_SERVER.ABNORMAL_START;
         write(3,3);
      end FAILURE;
   end loop;
end RECONFIGURE_1;


task ALTERNATE_CALLER is
   entry ABNORMAL_START;
   pragma distribute_to(2);
   pragma priority(1);
end ALTERNATE_CALLER;

task body ALTERNATE_CALLER is
begin
   write(4,1);
   accept ABNORMAL_START;
   loop
      write(4,2);
      SERVER.E;
      write(4,3);
   end loop;
end ALTERNATE_CALLER;
```

```
  task body SERVER is
  begin
     write(5,1);
     loop
        write(5,2);
        accept E;
        write(5,3);
     end loop;
  end SERVER;



  task RECONFIGURE_2 is
     entry FAILURE(WHICH : in integer);
     pragma distribute_to(2);
     pragma priority(2);
     for FAILURE use at 10;
  end RECONFIGURE_2;

  task body RECONFIGURE_2 is
  begin
     write(6,1);
     accept FAILURE(WHICH : in integer) do
        write(6,2);
        ALTERNATE_CALLER.ABNORMAL_START;
        write(6,3);
     end FAILURE;
  end RECONFIGURE_2;



begin
   null;
end;
```

APPENDIX 2

A NEW LINGUISTIC APPROACH TO BACKWARD ERROR RECOVERY[*]

Samuel T. Gregory John C. Knight[**]

Department of Computer Science
University of Virginia
Charlottesville, Virginia, U.S.A. 22903
(804) 924-7605

ABSTRACT

Issues involved in language facilities for backward error recovery in critical, real-time systems are examined. Previous proposals are found lacking. The dialog, a new building block for concurrent programs, and the colloquy, a new backward error recovery primitive, are introduced to remedy the situation. The previous proposals are shown to be special cases of the colloquy. Thus, the colloquy provides a general framework for describing backward error recovery in concurrent programs.

Subject Index:
Reliable Software – Interprocess Communication and Synchronization

[*]This work was sponsored by NASA grant number NAG1-260 and has been cleared for publication by the sponsoring organization.

[**]Presenter at FTCS-15 if paper is accepted.

1. INTRODUCTION

In this paper we examine the issues involved in the use of backward error recovery in critical, real–time systems. In particular, we are concerned with language facilities that allow programmers to specify how alternate algorithms are to be applied in the event that an error is detected. The best–known approach is the *conversation*[1]. Many difficulties with conversations have been pointed out including the lack of any time–out provision and the possibility of deserter processes. We introduce a new building block for concurrent programs called the *dialog* and a new backward–error–recovery primitive called the *colloquy* that remedy the various limitations of the conversation. The colloquy is constructed from dialogs and provides a general framework for describing backward error recovery in concurrent programs.

All of the syntactic proposals that we introduce are derived from Ada® [2]. The dialog and colloquy are proposed as general concepts but the specific syntax for their use is given as extensions to Ada. The actual syntax is irrelevant; the concepts could be used in many other programming languages. However, once chosen, a rigid syntax can allow a compiler to enforce certain of the semantic rules.

In section two, we briefly describe the concept of the conversation and the associated syntactic proposals that have been made. Issues that have been raised with conversations are discussed in section three. In section four, we present a syntax for the dialog called the *discuss* statement. In section five, we introduce the colloquy and a new statement called the *dialog_sequence* which allows the specification of the actions needed for a colloquy. In section six, we discuss the use of colloquys in the implementation of all previous approaches to backward error recovery.

®Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

2. CONVERSATIONS

The *conversation* is the canonical software fault–tolerance proposal for dealing with communicating processes. In a conversation a group of processes separately establish recovery points and begin communicating. At the end of their communication (i.e. the end of the conversation), which may include the passage of multiple distinct sets of information, they each wait for the others to arrive at an acceptance test for the group. If they pass the acceptance test, they *commit* to the information exchange that has transpired by discarding their recovery points and proceeding. Should they fail the acceptance test, they all restore their states from the recovery points. No process is allowed to *smuggle* information in or out by communicating with a process that is not participating in the conversation. Conversations can be nested; from the point of view of a surrounding conversation, a nested conversation is an atomic action[3].

Although not explicitly stated in the literature, it is assumed that if an error occurs during a conversation such that the acceptance test fails, the *same* set of conversant processes attempt to communicate again once individually rolled back and reconfigured (rather than proceeding on unrelated activities). It follows that they eventually reach the *same* acceptance test again. It is also presumed that any other failure of one of the processes is taken as equivalent to a failure of the acceptance test by all of them.

The processes in a conversation are the components of a system of processes. Error detection mechanisms for this system consist of announcement of failure by any one of the components and the single acceptance test. The acceptance test evaluates the combined states of the component processes with the designed intent of their communications. Damage assessment is complete *before* execution begins since the individual states of all the processes involved in the conversation are suspect, but no other processes are affected. Error recovery consists of restoring each process to the state it had as it entered the conversation, and the system of processes continues with its service by allowing each process to re–try the communication perhaps using an alternate mechanism within that process for the communication activity.

Conversations were originally proposed as a structuring or design concept without any syntax that might allow enforcement of the rules. Russell[4] has proposed the "Name-Linked Recovery Block" as a syntax for conversations. The syntax appropriates that of the recovery block[5]. What would otherwise be a recovery block, becomes part of a conversation designated by a conversation identifier. The primary and alternate activities of the recovery block become that process' primary and alternate activities during the conversation, and the recovery block's acceptance test becomes that portion of the conversation's acceptance test appropriate to this process. The conversation's acceptance test is evaluated after the last conversant reaches the end of its primary or alternate. If any of the processes fail its acceptance test, all conversants are rolled back.

Kim has examined several more possible syntaxes for conversations[6]. His approaches assume the use of monitors[7] as the method of communication among processes. He examines the situation from two philosophies toward grouping. In one scheme, the conversing activities are grouped with their respective processes' source code, but are well marked at those locations. In another scheme, the conversing actions of the several processes are grouped into one place so that the conversation has a single location in the source code. The issue he is addressing is whether it is better to group the text of a conversation and scatter the text of a process or to group the text of a process and scatter the text of a conversation. A third scheme attempts to resolve the differences between the first two.

3. ISSUES WITH CONVERSATIONS

Desertion is the failure of a process to enter a conversation or arrive at the acceptance test when other processes expect its presence. Whether the process will never enter the conversation, is simply late, or enters the conversation only to take too long or never arrive at the acceptance test, does not matter to the others if they have real–time deadlines to meet. Each process may have its own view of how long it is willing to wait, especially since processes may enter a conversation asynchronously. Whether they protect inter–process communications or sequential parts of processes, acceptance tests must be reached and reached on time for the results to be useful. Meeting real–time deadlines is as important to providing the specified service as is producing correct output. In order to deal effectively with desertion, especially in critical systems, some form of timing specification on communication and on sequential codes is vital.

When it needs to communicate, a process enters a conversation and stays there, perhaps through many alternate algorithms, until the communication is completed successfully. The same group of processes are required to be in the alternate interactions as were in the primary. The recovery action merely sets up the communication situation again. In the original form of conversation, once a process enters the construct, it cannot break out and *must* continue trying with the same set of other processes, including one or more which may be incapable of correct operation. In practice, when a process fails in a primary attempt at communication with *one group of processes* to achieve its goal, it may want to attempt to communicate with an *entirely different group* as an alternate strategy for achieving that goal; in fact, different processes might make different numbers of attempts at communicating. Conversations do not allow this, although it is not desertion if it is systematic and intended.

In a conversation, once individually rolled back and reconfigured, the same set of conversant processes attempt to communicate again, and eventually reach the *same* acceptance test again. True *independence* of algorithms between primary and alternates, within the context of backward error recovery, might require very *different* acceptance tests for each algorithm, particularly if some of them provide significantly degraded services. A single test for achievement of a process' goal at a particular point in its text would of necessity have to be general enough to pass results of the most degraded algorithm. This might be too general to enable it to catch errors produced by other, more strict, algorithms. These considerations suggest the need for separate acceptance tests specifically tailored for each of the primary and alternate algorithms.

It must also be remembered, that although each process has its own reasons for participating, there is a goal for the *group* of processes as well. Rather than combine the individual goals of the many participants with the group goal in a single acceptance test (perhaps allowing the programmer to forget some), and rather than replicating the test for achievement of the group goal within every participant, there should be a separate acceptance test for each participant and another for the group.

A final problem with the conversation concept as it was originally defined, is that if a process runs out of alternates, no scheme is provided or mentioned for dealing with the situation.

4. THE DIALOG

We define a *dialog* to be an occurrence in which a set of processes:

(a) establish individual recovery points,

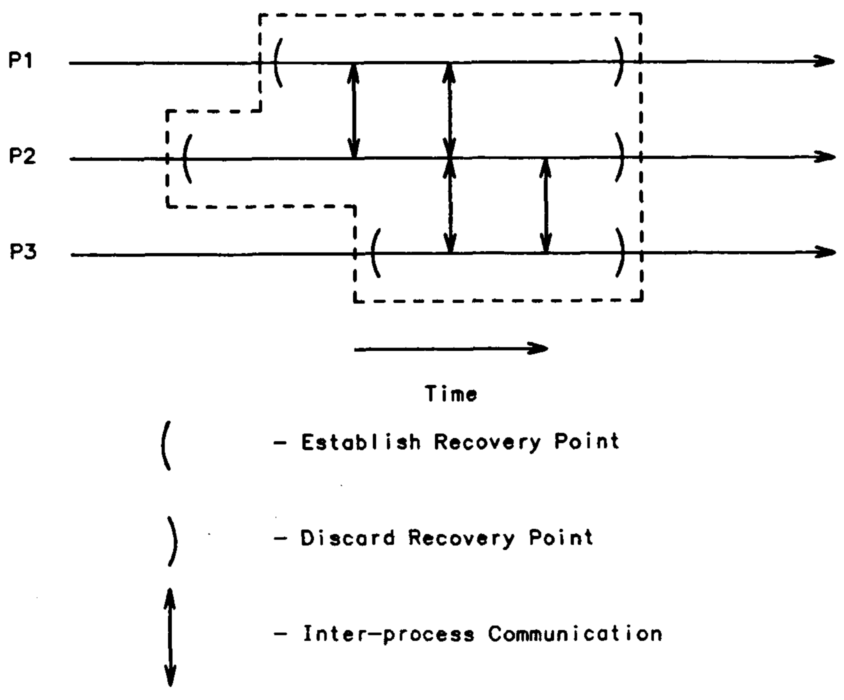
(b) communicate among themselves and with no others,

(c) determine whether all should discard their recovery points and proceed or restore their states from their recovery points and proceed, and

(d) follow this determination.

Success of a dialog is the determination that all participating processes should discard their recovery points and proceed. *Failure* of a dialog is the determination that they should restore their states from their recovery points and proceed. Nothing is said about what should happen *after* success or failure; in either case the dialog is complete. Dialogs may be properly nested, in which case the set of processes participating in an inner dialog is a subset of those participating in the outer dialog. Success or failure of an inner dialog does not necessarily imply success or failure of the outer dialog. Figure 1 shows a set of three processes communicating within a dialog.

We introduce the *discuss* statement as a syntactic form that can be used to denote a dialog. Figure 2 shows the general form of a discuss statement. The *dialog_name* associates a particular discuss statement with the discuss statements of the other processes participating in this dialog, *dynamically* determining the constituents of the dialog. This association cannot in general be known statically. At execution time, when control enters a


P1
P2
P3
Time
(- Establish Recovery Point
) - Discard Recovery Point
- Inter-process Communication

Three Processes Communicating in a Dialog
Figure 1

```
DISCUSS dialog_name BY

   sequence_of_statements

TO ARRANGE Boolean_expression;
```

A DISCUSS Statement
Figure 2

process' discuss statement with a given dialog name, that process becomes a participant in a dialog. Other participants are any other processes which have already likewise entered discuss statements with the same dialog name and have not yet left, and any other processes which enter discuss statements with the same dialog name before this process leaves the dialog. Either all participants in a dialog leave it with their respective discuss statements successful, or all leave with them failed, i.e. the dialog succeeds or fails.

The *sequence of statements* in the discuss statement represent the actions which are this process' part of the group's actions within their dialog. Any inter-process communication *must take place within* this sequence of statements (i.e. be protected by a dialog). The discuss statement fails if an exception is raised within it, if an enclosed *dialog_sequence* (see below) fails, or if any timing constraint is violated.

The *Boolean_expression* is an acceptance test on the results of executing the sequence of statements. It represents the process' *local* goal for the interactions in the dialog. It is evaluated after execution of the sequence of statements. If this Boolean expression or that in the corresponding discuss statement of any other process participating in this dialog is evaluated **false**, the discuss statement of each participant in the dialog *fails*. If all of the local acceptance tests succeed, the common goal of the group, i.e. the *global* acceptance test is evaluated. If this common goal is **true**, the corresponding discuss statements of all participants in the dialog succeed; otherwise they fail. Syntactically, the common goal is specified by a parameterless Boolean function with the same name as the dialog name in the discuss statement.

We stated that the participants in a particular dialog cannot be known statically. There may be, say, three processes whose texts contain references to a particular dialog name. If two of them enter a dialog using that name, questions might arise about participation of the third. The third process may be executing some other portion of its code so that it is unlikely to enter a dialog of that name in the near future. If the two processes reach and pass their acceptance tests, they, being the only participants in the dialog, can leave it -- the third process is not necessary to the dialog, so is not a deserter. If the dialog fails due to an acceptance test or a timeout (see below), the problem is not guaranteed to be the absence of the third process, so again it is not (necessarily) a deserter. If the dialog has no time limit specified (see below), that had to be by conscious effort of the programmer, so the two processes becoming "hung" in the dialog while waiting for the third was *not* unplanned.

The dialog names used in discuss statements are required to be declared in *dialog declarations*. The general form of a dialog declaration is:

```
DIALOG function_name SHARES ( name_list );
```

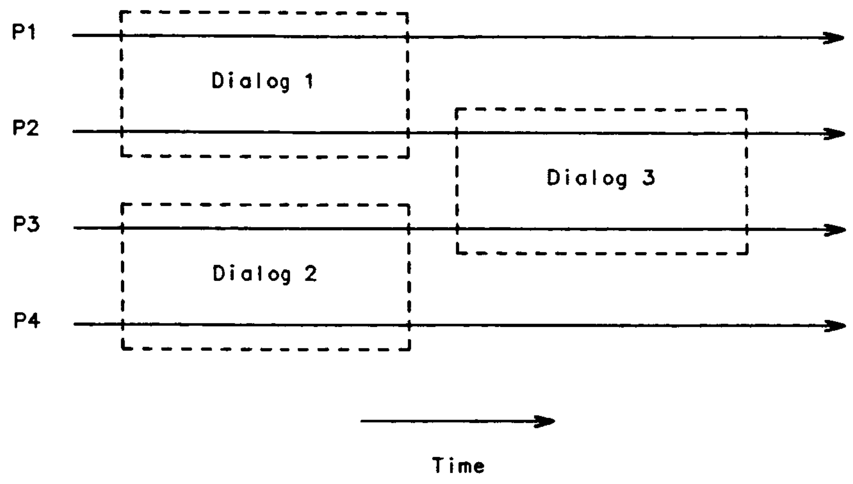
The *function_name* is the identifier being declared as a dialog name (and the name of the function defining the global acceptance test). The names mentioned in the *name_list* are the names of *shared* variables which will be used within dialogs that use this dialog name. This includes variables used within the function that implements the global acceptance test. Only a variable so named may be used within a discuss statement, and then only within discuss statements using a dialog name with that variable's name in its dialog declaration. The significance of these rules is that the set of shared variables can be locked by the compiler and execution-time support system to prevent smuggling. In effect, the actions of the dialog's participants are made to appear atomic to other processes with respect to these variables. (Our implementation, not described here, also prevents smuggling via messages or rendezvous).

The Boolean function named by the dialog name is evaluated after all processes in the dialog have evaluated their respective Boolean expressions and they all evaluate to **true.** It is only evaluated once for an instance of the dialog; i.e. it is not evaluated by each participating process. Thus no process can leave a dialog until all processes currently in that dialog leave with the same success, and success involves the execution of both a local and a global acceptance test.

5. THE COLLOQUY

A *colloquy* is a semantic construct that solves the problems of conversations. Unlike conversations, the rules of order and participation are well-defined and explicitly laid out.

A colloquy is a collection of dialogs. At execution time, a dialog is an interaction among processes. Each individual process has its own *local* goal for participating in a dialog, but the group has a larger *global* goal; usually providing some part of the service required of the entire system. If, for whatever reason, any of the local goals or the global goal is not achieved, a backward error recovery strategy calls for the actions of the particular dialog to be undone. In attempting to ensure continued service from the system, each process may make another attempt at achieving its original local goal, or some *modified* local goal through entry into a *different* dialog. Each of the former participants of the now defunct dialog may choose to interact with an *entirely separate* group of processes for its alternate algorithm. The altered constituency of the new dialog(s) most certainly requires new statement(s) of the original global goal. The set of dialogs which take place during these efforts on the processes' part is a *colloquy.* A set of four processes engaged in a colloquy that involves three dialogs is shown in Figure 3.



Four Processes in a Colloquy of Three Dialogs
Figure 3

A colloquy, like a dialog or a rendezvous in Ada, does not exist syntactically but is entirely an execution–time concept. The places where the text of a process statically announces entry into colloquys are marked by a variant of the Ada select statement called a *dialog_sequence.*

The general form of a dialog_sequence is shown in Figure 4. At execution time, when control reaches the **select** keyword, a recovery point is established for that process. The process then *attempts* to perform the activities represented in Figure 4 by `attempt_1`. The attempt is actually a discuss statement followed by a sequence of statements. To ensure proper nesting of dialogs and colloquys, a discuss statement may appear only in this context. If the performance of these activities is *successful,* control continues with the statements following the dialog_sequence. The term "success" here means that no defensive, acceptability, or timing checks occurring within the attempt detected an error, and that no exceptions (if the language has exceptions) were propagated out to the attempt's discuss statement. If the attempt was not successful, the process' state is restored from the

```
SELECT
      attempt_1
   OR
      attempt_2
   OR
      attempt_3

   TIMEOUT simple_expression
      sequence _of_statements

   ELSE
      sequence _of_statements
END SELECT;
```

Dialog_Sequence
Figure 4

recovery point and the other attempts will be tried in order. Thus, the dialog_sequence enables the programmer to provide a primary and a list of alternate algorithms by which the process may achieve its goals at that locus of its text.

Exhaustion of all attempts with no success brings control to the **else** part after restoration of the process' state from the recovery point. The **else** part contains a sequence of statements which allows the programming of a "last ditch" algorithm for the process to achieve its goal. If this sequence of statements is successful, control continues after the dialog_sequence. If not, or if there was no statement sequence, the surrounding attempt fails.

Timing constraints can be imposed on colloquys (and hence on dialogs). Any participant in a colloquy can specify a timing constraint which consists of a simple expression on the **timeout** part of the dialog_sequence. Absence of a timing constraint must be made explicit by replacing the simple expression with the keyword **never**. A timing constraint specifies an interval during which the process may execute as many of the attempts as necessary to achieve success in one of them. Should an attempt achieve success or the list of attempts be exhausted without success before expiration of the interval, further actions are the same as for dialog_sequences without timing specifications. However, if the interval expires, the current attempt fails, the process' state is restored from the recovery point, and execution continues at the sequence of statements in the **timeout** part. The attempts of the other processes in the same dialog also fail but their subsequent actions are determined by their own dialog_sequences. If several participants in a particular colloquy have timing constraints, expiration of one has no effect on the other timing constraints. The various intervals expire in chronological order. As with the else part, the timeout part allows the programming of a "last ditch" algorithm for the process to achieve its goal, and is really a form of forward recovery since its effects will not be undone, at least at this level. If the sequence of statements in the timeout part is successful, control continues after the dialog_sequence. If not, or if there were no statement sequence, the surrounding attempt fails.

In any attempt, a statement sequence (which is logically outside the dialog_sequence) can follow the discuss statement to provide specialized post-processing after the recovery point is discarded if the attempt succeeds. It is not subject to this dialog_sequence's timing constraint.

The programmer is reminded by its position after the **timeout** part that the **else** part is not protected by the timer, and that it is reached only after other (potentially time-consuming) activities have taken place. The structure of the dialog_sequence also requires no acceptance check on these activities. The implication of these two observations is that the last ditch activities need to be programmed very carefully.

A *fail* statement may occur only within a sequence of statements contained within a dialog_sequence. Execution of a fail statement causes the encompassing attempt to fail. The fail statement is intended for checking within an attempt. For example, it can be used to program explicit defensive checks on inputs such as:

```
IF input_variable < lower_bound THEN
   FAIL;
END IF;
```

It can also be used to simplify the logical paths out of an attempt should some internal case analysis reach an "impossible" path. With the fail statement, the programmer does not have to make the code for the attempt complicated by providing jumps or other paths to the acceptance test or to insure that some part of the test is always **false** for such a special path. The fail statement can also be used to provide sequences of statements for the **else** and **timeout** parts that make failure explicit rather than implicit (i.e. failure is indicated by their *absence*).

6. OTHER LANGUAGE FACILITIES

Dialog_sequences can be used to construct deadlines[8], generalized exception handlers[9], recovery blocks, traditional conversations, exchanges[10], and s–conversations[11]. Thus the colloquy is at least as powerful as each of these previously proposed constructs for provision of fault tolerance. For the sake of brevity, we will illustrate only the programming of a recovery block.

A recovery block is a special case of a colloquy in which there is only one process participating, every dialog uses the same acceptance test, there is no timing requirement, and there are no "last ditch" algorithms to prevent propagation of failures of the construct. Figure 5 shows a dialog_sequence that is equivalent to the recovery block shown in Figure 6. The use of the fail statement in the dialog_sequence makes explicit the propagation of the error to a surrounding context just as does the **else error** closing of the recovery block. In the dialog_sequence, the Boolean expression is repeated in the discuss statements rather than gathered into the dialog function because we want to be able to include local variables in it as a programmer of the recovery block would. Should an error be detected in `statement_sequence_1`, the state is restored and `statement_sequence_2` is executed, and so on.

```
FUNCTION abc RETURNS boolean IS BEGIN RETURN TRUE; END abc;
   ....
DIALOG abc SHARES ( );
   ....

SELECT
      DISCUSS abc BY
         statement_sequence_1
      TO ARRANGE boolean_expression_1;

   OR
      DISCUSS abc BY
         statement_sequence_2
      TO ARRANGE boolean_expression_1;

   OR
      DISCUSS abc BY
         statement_sequence_3
      TO ARRANGE boolean_expression_1;

   TIMEOUT NEVER;

   ELSE
       FAIL; — Omitting this line does not change the semantics.
END SELECT;
```

Specification of Colloquy for a Recovery Block
Figure 5

```
ENSURE boolean_expression_1 BY
   statement_sequence_1

ELSE BY
   statement_sequence_2

ELSE BY
   statement_sequence_3

ELSE ERROR;
```

A Recovery Block
Figure 6

Finally, should an error be detected in `statement_sequence_3`, the state is restored and the error is signaled in a surrounding context. An error may be detected by evaluation of boolean_expression_1 to **false**, or by violation of some underlying interface (such as raising of an exception).

7. CONCLUSIONS

We have introduced a new linguistic construct, the colloquy, which solves the problems identified in the earlier proposal, the conversation. We have shown that the colloquy is at least as powerful as recovery blocks, but it is also as powerful as all the other language facilities proposed for other situations requiring backward error recovery; recovery blocks, deadlines, generalized exception handlers, traditional conversations, s-conversations, and exchanges.

The major features that distinguish the colloquy are:

(1) The inclusion of explicit and general timing constraints. This allows processes to protect themselves against any difficulties in communication that might prevent them from meeting real-time deadlines. It also effectively deals with the problem of deserter processes.

(2) The use of a two-level acceptance test. This allows much more powerful error detection because it allows the tailoring of acceptance tests to specific needs.

(3) The reversal of the order of priority of alternate communication attempts and of recovery points. This allows processes to choose the participants in any alternate algorithms rather than being required to deal with a single set of processes.

(4) A complete and consistent syntax that is presented as extensions to Ada but could be modified and included in any suitable programming language.

Sample programs that have been written (but not executed) using the colloquy show that extensive backward error recovery can be included in these programs simply and elegantly. We are presently implementing these ideas in an experimental Ada testbed.

This paper is not a formal statement of these concepts. The reader may correctly feel that important detail has been omitted. We are only able to present informally the key concepts in a paper of this length. For more details, see [12].

8. ACKNOWLEDGEMENTS

This work was sponsored by NASA grant number NAG1–260 and has been cleared for publication by the sponsoring organization.

REFERENCES

(1) Randell B., "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, **SE-1**(2), pp. 220–232, June 1975.

(2) *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, 22 January 1983.

(3) Lomet D.B., "Process Structuring, Synchronization and Recovery Using Atomic Actions," *SIGPLAN Notices*, **12**(3), pp. 128–137, March 1977.

(4) Russell D.L., M.J. Tiedeman, "Multiprocess Recovery Using Conversations," *Digest of Papers FTCS-9: Ninth Annual Symposium on Fault-Tolerant Computing*, p. 106, June 1979.

(5) Horning J.J., et al., "A Program Structure for Error Detection and Recovery," pp. 171–187 in *Lecture Notes in Computer Science* **Vol. 16**, ed. E. Gelenbe and C. Kaiser, Springer-Verlag, Berlin, 1974.

(6) Kim K.H., "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Transactions on Software Engineering*, **SE-8**(3), pp. 189–197, May 1982.

(7) Per Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

(8) Campbell R.H., K.H. Horton, G.G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," *Digest of Papers FTCS-9: Ninth Annual Symposium on Fault-Tolerant Computing*, pp. 95–101, 1979.

(9) Salzman E.J., *An Experiment in Producing Highly Reliable Software,* M.Sc. Dissertation, Computing Laboratory, University of Newcastle upon Tyne, 1978.

(10) Anderson T., J.C. Knight, "A Framework for Software Fault Tolerance in Real–Time Systems," *IEEE Transactions on Software Engineering*, **SE–9**(3), pp. 355–364, May 1983.

(11) Jalote P., R.H. Campbell, "Fault Tolerance Using Communicating Sequential Processes," *Digest of Papers FTCS–14: Fourteenth International Conference on Fault–Tolerant Computing*, pp. 347–352, 1984.

(12) Gregory S.T., *Programming Language Facilities for Comprehensive Software Fault–Tolerance in Distributed Real–Time Systems*, Ph.D. Dissertation, Department of Computer Science, University of Virginia, 1985.

DISTRIBUTION LIST

Copy No.	
1 - 3	National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665 Attention: Mr. Edmond H. Senn ACD, Computer Science and Applications Board
4 - 5*	NASA Scientific and Technical Information Facility P. O. Box 8757 Baltimore/Washington International Airport Baltimore, MD 21240
6 - 7	J. C. Knight
8	A. C. Weaver
9 - 10	E. H. Pancake Clark Hall
11	SEAS files

*One reproducible copy

JO# 5736 CMS